MIT/LCS/TR-517

[redacted]

[redacted]

September 1991

This blank page was inserted to preserve pagination.

Algorithms for Search Trees on Message-Passing Architectures

by

Adrian Colbrook
Eric A. Brewer
Chrysanthos N. Dellarocas
William E. Weihl

September 1991

Abstract

In this paper we describe a new algorithm for maintaining a balanced search tree on a message-passing MIMD architecture; the algorithm is particularly well suited for implementation on a small number of processors. We introduce a $(2^{B-2}, 2^B)$ search tree that uses a linear array of $\mathcal{O}(\log n)$ processors to store n entries. Update operations use a bottom-up node-splitting scheme, which performs better than top-down search tree algorithms. Additionally, for a given cost ratio of computation to communication the value of B may be varied to maximize performance. Implementations on a parallel-architecture simulator are described.

Keywords: Balanced search trees, Parallel algorithms, Linear processor array, Message-passing architectures.

This work was supported in part by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and by an equipment grant from Digital Equipment Corporation. Adrian Colbrook was supported by a Science and Engineering Research Council Postdoctoral Fellowship, Eric A. Brewer by a Office of Naval Research Fellowship, and Chrysanthos N. Dellarocas by a Starr Foundation Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

1 Introduction

We introduce a new balanced search tree algorithm for message-passing architectures. The algorithm assumes a linear array of processors each with a large local memory; such arrays are easily emulated on most message-passing MIMD architectures. The algorithm performs updates using a bottom-up node-splitting scheme and shows improvements in throughput and response time when compared to similar top-down algorithms [GS78, CT84].

Search trees are widely used for fast implementations of dictionary abstract data types. A dictionary is a partial mapping from keys to data that supports three operations: *insert*, *delete* and *search*. For simplicity we will assume that the dictionary stores no data with the keys and so may be viewed as a set of keys. A number of useful computations can be implemented in terms of dictionary abstract data types, including symbol tables, priority queues and pattern-matching systems.

The B-tree was originally introduced by Bayer [Bay72]. The B-tree algorithms for sequential applications were designed to minimize the response time for a single query and the sequential algorithm for a single search operation on a balanced B-tree has logarithmic complexity. The improvement in the response time that can be achieved by a parallel algorithm for a single search can at best be logarithmic in the number of processors [Qui87]. Thus, for parallel systems a more important concern is the system throughput for a series of search, insertion and deletion operations executing in parallel.

We introduce the $(2^{B-2}, 2^B)$[1] search tree, a variation of the B-tree. A $(2^{B-2}, 2^B)$ search tree (for $B \geq 3$) is a tree in which every branch node (except the root) has between 2^{B-2} and 2^B children, and every path from the root to a leaf has the same length. For example, B=*3* gives a $(2, 8)$ tree, B=*4* gives a $(4, 16)$ tree and so on. The root has between 2 and 2^B children. Only leaf nodes store key values; branch nodes store index information used to find the appropriate leaf node. A leaf node stores between 2^{B-2} and 2^B keys. Each leaf node stores key values within a contiguous range; the ranges of all leaf nodes partition the set of possible key values and are in ascending order from the leftmost node to the rightmost. The index information stored at a non-leaf node is simply the lowest key value associated with each of its children. Thus the set of key values is partitioned at every level in the tree.

A search tree of n entries is implemented on an array of up to $\log_2 n/(B-2)+1$ processors. Each processor holds a level of the tree structure in local memory and the last processor stores the actual keys. Therefore, the memory required to store the search tree increases by a factor of 2^B between adjacent processors down the linear array. Figure 1 shows this configuration for a $(2, 8)$ tree. Processor P_1 stores the leaf nodes of the tree. Operations are invoked by requests entering the array at the top; the replies these generate leave from the bottom. Each processor communicates only with its immediate neighbors.

[1] This is pronounced, "two to the B minus two, two to the B"; or simply "two B".

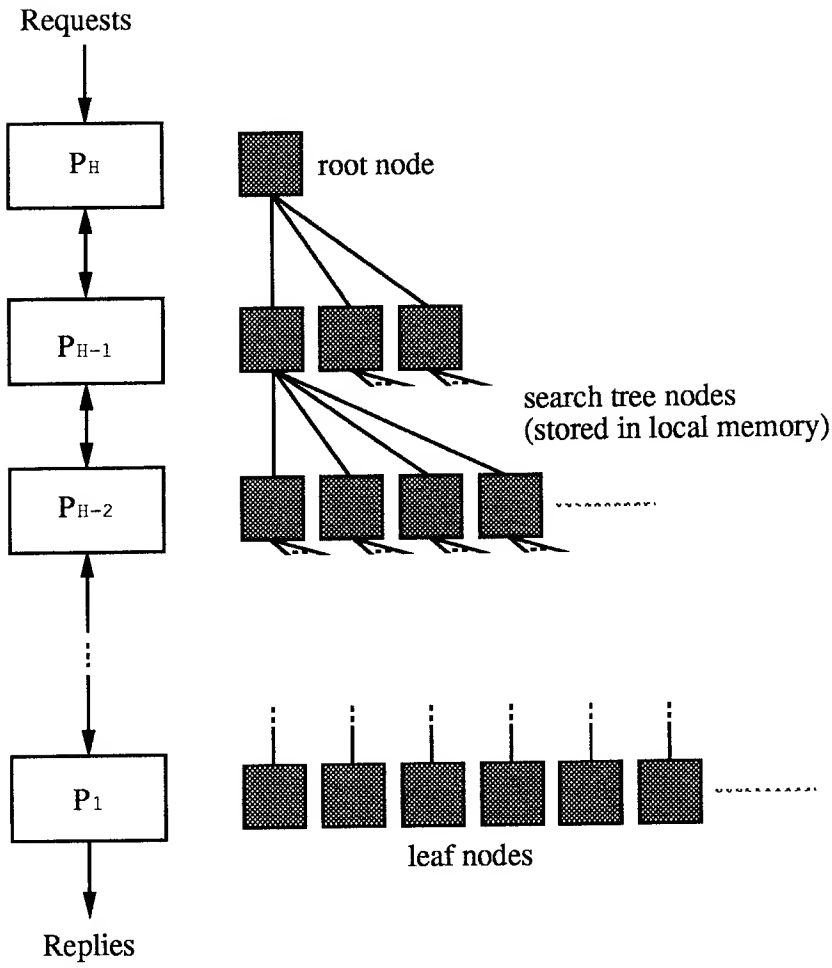


Figure 1: The processor configuration for a $(2, 8)$ tree of height H.

In our new algorithm a dictionary is represented by a modified $(2^{B-2}, 2^B)$ search tree in which links are added from each node to its left and right neighbors in the same level. This is similar to the algorithm developed by Weihl and Wang [WW90, Wan91] for B-trees, which in turn was based on an algorithm developed by Lehman and Yao [LY81] and modified by Sagiv [Sag86]. In these *link method* algorithms [SG88] right links are added to adjacent nodes in a B-tree. In the new algorithm both left and right links are added between adjacent siblings.

Carey and Thompson [CT84] implemented a 2-3-4 search tree using a linear array of $\mathcal{O}(\log n)$ processors. A *2-3-4 tree* is a tree in which each node that is not a leaf has two, three or four children, and every path from the root to a leaf is the same length. The scheme allows update operations to be performed using the top-down node-splitting scheme presented by Guibas and Sedgewick [GS78]. Mond and Raz [MR85] also proposed a top-down strategy for B-trees. A linear array of processors was used

by Tanaka, Nozaka and Masuyama [TNM80] in their pipelined binary-tree algorithm, and Fisher [Fis84] proposed a pipeline system that used a pipeline length proportional to the length of the key.

We first implemented a $(2^{B-2}, 2^B)$ search tree using a top-down node-splitting scheme. We then implemented the search tree using the bottom-up algorithm. Both of these algorithms have been implemented using Proteus [Del91, Bre91, BDCW91], a multiprocessor simulator developed at MIT. We measured the throughput and response times for various processor-array lengths, tree-branching factors, query mixes, message-passing paradigms and one-way message delays. From these measurements we compared the performance of the algorithms and determined the optimal tree structures. The bottom-up algorithm has better throughput and response time than the top-down algorithm in every case.

In Section 2 we present the issues that must be addressed by concurrent search tree algorithms and motivate the development of the new algorithm. Sections 3 and 4 describe the top-down and bottom-up algorithms. Section 5 describes the Proteus simulator and Section 6 outlines the implementation of the algorithms using Proteus. Section 7 presents the performance evaluation of both algorithms; Section 8 describes design alternatives for the bottom-up algorithm. Finally, we present our conclusions in Section 9.

2 Concurrent Search Tree Algorithms

The large number of concurrent search tree algorithms presented in the literature prevents a complete description of each in this paper. Instead, we discuss the common issues that are addressed by all of the algorithms. Much of this discussion is based upon Wang's analysis of concurrent search tree algorithms [Wan91].

All concurrent search tree algorithms share the problem of managing contention. Concurrency control is required to ensure that two or more independent processes accessing a B-tree do not interfere. A common approach is to associate a read/write lock with every node in the search tree [LSS87]. This causes data contention as writers block incoming writers and readers, and readers block incoming writers. The contention is severe when it occurs at higher levels in the search tree, particularly at the root, which is often termed a *root bottleneck*.

Similar problems are caused by resource contention. In a shared-memory architecture all of the processes trying to access the same tree node will access the same memory module on the machine. Similarly, for message-passing architectures, the processor on which a node resides will receive messages from every processor trying to access the node. Resource contention is again most serious for the higher levels in the search tree. Node replication [Wan91] reduces contention but requires a coherence protocol to maintain consistency.

Associated with the contention issue is the problem of *process overtaking*. This may occur when a process that holds a lock selects the next node it wishes to access, releases its lock and attempts to acquire a lock for the next node. A second process may acquire a lock on the next node between the original process releasing the old lock and acquiring the lock for the next node. The second process can then update the node in such a fashion as to cause the first process to lock the wrong node when it eventually acquires the lock. To prevent this kind of process overtaking many algorithms have their operations use *lock coupling* to block independent operations. An operation traverses the tree by obtaining the appropriate lock on the child before releasing the lock it holds on the parent. This technique is used in the top-down algorithms [GS78, CT84, MR85, CS90]. The link method algorithms eliminate the need for lock coupling. If the wrong node is reached at any stage the side links are traversed until the correct node is found. This reduces the number of locks that must be held concurrently and increases throughput. However, traversing the links could in theory lead to an increase in the response time.

Linear arrays of processors provide a processor-efficient means of implementing search tree algorithms. Since each level of the search tree is stored in the local memory of a single processor, the contention for resources is approximately uniform throughout the structure. Our algorithm uses this structure and is very simple. The code that implements a level of the structure is replicated on all the processors in the array, with some minor modifications for the processor storing the leaf nodes. Side links avoid the need for lock coupling, which results in higher concurrency. The bottom-up algorithm presented in this paper leads to significant improvements in throughput over the top-down algorithms. In addition, we show that the response time for the new algorithm is also less than that for top-down algorithms. Our results show that the bottom-up algorithm described here has the best performance of any of the implementations of search trees for linear arrays described to date.

3 The Top-Down Algorithm

The top-down algorithm allows insertions, deletions and exact-match searches on a $(2^{B-2}, 2^B)$ search tree. The *search* operation is a simple version of the normal B-tree search operation [Com79]. The *insert* and *delete* operations are based upon the top-down node-splitting scheme introduced by Guibas and Sedgewick [GS78], in which transformations are applied during a single traversal of the tree. The tree is traversed from the root downward and transformations are applied between adjacent levels at the same time.

The *insert* operation performs node splitting upon encountering a 2^B-branch node, other than the root. A new right brother of the node is created and the branches of the original node are divided, forming two 2^{B-1}-branch nodes. Figure 2 shows an example of this transformation applied to a $(2, 8)$

tree. This transformation ensures that any future node splitting does not cause upward propagation in the tree structure; this allows the transformation to be applied in a top-down fashion. This follows by induction on the depth of the tree, since any transformation is applied to the parent of a node before being applied to the node itself. Therefore, the parent must have a degree of at most $2^B - 1$ when a transformation is applied to the node.

When a *delete* operation encounters a 2^{B-2}-branch node, other than the root, one of two deletion transformations is applied. If a neighboring node has less than or equal to 2^{B-1} branches, the node and its neighbor are merged to form a single node. Otherwise the branches of the node and its neighbor are redistributed evenly between the two nodes. Figures 3 and 4 show examples of these transformations applied to a $(2, 8)$ tree. The neighbor relationship used in the deletion algorithm relates a node to its right brother in the subtree, or in the case of the rightmost node, to its left brother. These transformations ensure that any future merging of nodes will not cause upward propagation of transformations.

Several B-tree algorithms perform delete restructuring only when a node is empty [Wan91]. These strategies reduce the probability that the nodes need to be merged, thus reducing the amount of work required. However, merging nodes when they are one quarter full preserves efficient space utilization, and bounds the height of the tree and therefore the number of processors required. This is particularly important for linear array implementations, where efficient space utilization is required and the number of available processors may be limited.

When the transformations are applied to a root node, the insertion transformation converts a root node with 2^B branches into a double 2^{B-1}-branch node configuration and a new root node, increasing the height of the tree. The merging deletion transformation converts a root node with 2 branches into a new root node formed by the merging of the root's children thus reducing the height of the tree. The redistribution deletion transformation does not lead to a reduction in the height of the tree.

A further property of the $(2^{B-2}, 2^B)$ tree is worth noting at this point. The insertion and deletion transformations of some B-tree algorithms [CT84, Com79] are direct inverses. In these algorithms an application of a transformation immediately followed by its inverse can occur and this can result in oscillation. For example, in the 2-3-4 tree algorithm described by Guibas and Sedgewick [GS78], an insertion transformation splits a 4-branch node into two 2-branch nodes. If a deletion operation is then applied to the original node, the two 2-branch nodes are merged reforming the original 4-branch node. This allows one transformation per operation. This problem is most severe in higher levels of the search tree where updates are less frequent. Performing these transformations increases the average response time and reduces the throughput of the system. A stabilizing (hysteresis) effect occurs in the $(2^{B-2}, 2^B)$ tree since the insertion transformation leaves each node (except the root) with 2^{B-1} children.

Before a deletion transformation may be applied to the node, 2^{B-2} children must be removed from it. [2] Therefore, the oscillations encountered in other algorithms cannot occur. The root node exhibits similar hysteresis behavior. When the tree grows a new root node is created with two children. Each of these nodes has 2^{B-1} branches. Before the tree can shrink, 2^{B-2} children must be removed from one of these nodes.

The implementation of the top-down algorithm on a reconfigurable system of transputers [Inm86] is described in Colbrook and Smythe [CS90]. It is worth noting that the *Proteus* simulator gave comparable results to those reported by Colbrook and Smythe.

4 The Bottom-Up Algorithm

In the bottom-up algorithm both left and right links are added between adjacent siblings in a $(2^{B-2}, 2^B)$ search tree. These links provide an additional method of reaching a node. The intent of this scheme is to make all nodes in a single level reachable from any other node at that level. If the wrong node is selected at the level above then the correct node can be found by using the links. This allows for an efficient solution to the process overtaking problem and permits changes to the tree structure to be made by a background task.

The algorithm again allows insertions, deletions and exact-match searches on a $(2^{B-2}, 2^B)$ search tree. Each of these operations begins by calling a *find* operation, which traverses the tree from the root until it reaches the leaf node that may store the specified key. For a *search* operation the keys stored at the leaf node are then searched for the specified key and the result is sent to the inquiring process.

An *insert* operation attempts to add the specified key to the keys stored at the leaf. If the leaf already stores 2^B keys and the specified key is not one of these, then an insertion transformation is applied causing the leaf node to be split into two 2^{B-1}-key nodes. The new node becomes the right brother of the original leaf node. The specified key is then added to the keys stored at the appropriate node and the *insert* operation returns. The work required to propagate the split to higher levels is carried out as a background task. This task adds a pointer to the newly created node at the appropriate node in the next level. This in turn may cause an insertion transformation: the split is propagated until a level is reached where no split occurs. Should the root of the tree be split then a new root is created and the height of the tree increases by one. Figure 5 shows an example of the insertion transformation applied to a $(2, 8)$ tree.

A *delete* operation attempts to remove the specified key from the keys stored at the leaf. If the leaf stores 2^{B-2} keys, one of which is the specified key, then one of two transformations is applied to the node

[2] Since $B \geq 3 \Rightarrow 2^{B-2} \geq 2$.

and its right neighbor in the tree (or the left neighbor for the rightmost leaf). These transformations are identical to those used for the top-down algorithm. The *delete* operation then returns and the propagation of the transformation to higher levels is again carried out as a background task. In this case there is no guarantee that the node and its neighbor share the same parent at the higher level, so the transformation may cause the index values associated with the nodes at a higher level to change. Transformations and changes to the index values are propagated until no further changes are required at a higher level. Should the nodes at the level below the root be merged to form a single node then this node becomes the root and the height of the tree decreases. Figures 6 and 7 show examples of these transformations applied to a $(2, 8)$ tree.

Since the changes to non-leaf levels caused by a transformation are carried out as a background task, the *find* operation must guarantee that the correct leaf node is reached even if the tree traversal is made between a transformation at a leaf node and the completion of the subsequent transformations and changes at higher levels. To achieve this we introduce a notion of *covers* for each node. An index node has associated with it a value i, the minimum key value that may be stored in the subtree rooted at the node. A key k is *covered* by a node x if and only if x*'s* index label and the index label of x's right neighbor, y, indicate that the leaf that may store k is a descendant of x. That is, $covers(k, x) \Leftrightarrow x.i \leq k < y.i$. When x is the rightmost node at a given level, $covers(k, x) \Rightarrow x.i \leq k$. When a *find* operation encounters a non-leaf node that does not cover the specified key, then the level is traversed from the node to either the left (if $x.i > k$) or to the right (if $y.i \leq k$) until the node that covers k is found.

In the case of a deletion transformation that causes two nodes to be merged, the neighbor is not removed immediately but is flagged as deleted and the links pointing to it from other nodes at the same level are updated. This allows references to a deleted node to be made by *find* until the result of the transformation has propagated to the higher level. When the *find* operation encounters a deleted node the traversal immediately begins at the left brother of the deleted node.

5 The Simulator

The Proteus system simulates the events that take place in a parallel machine at the level of individual machine instructions. The user writes a parallel program using a simple superset of the C programming language and a set of supported simulator calls. The parts of the user program executed locally on each processor are written in standard C and translated by the C compiler into machine code for the computer running the simulation. All nonlocal interactions, such as message passing, are performed by the supported simulator calls, which correspond to the machine code instructions that perform nonlocal interactions in real parallel machines.

We simulate a multiprocessor configured as a bidirectional ring. Each node consists of a processor, local memory, and a network chip for routing messages without using processor cycles. The network is packet switched and uses wormhole routing. We assume that a message fits in one packet. Wormhole routing (as opposed to store-and-forward) is relevant only for messages that travel more than one hop. We would expect a store-and-forward network to produce similar throughput results, but have worse response times. The latter results from the intermediate-node delays on messages to and from the processor used to send queries to the root node and receive replies from leaf and root nodes.

The one-way delay for a message is the sum of several values: the time to put a message on the network, the delay across the network, and the delay at the target. We assume a minimum wire delay of one cycle per (4-byte) word and a minimum switch delay of one cycle per word. Without contention a five-word message (the typical message size for both algorithms) requires seven cycles to go one hop: one cycle each for the source processor, the wire, and the target processor for the first word and an additional cycle for each of the subsequent words. We discuss the effect of longer message delays in Section 7.

6 Implementation

The top-down and bottom-up algorithms have been implemented on Proteus. A designated processor, termed the *Server*, is used to send queries to and receive replies from the processor storing the root node. The *Server* also receives the replies generated by the queries from the processor storing the leaf nodes.

In this section the implementation of the *insert*, *delete* and *search* operations for each of the algorithms is described. For a tree of H levels, processor P_h (where $1 < h \leq H$) is an index processor and processor P_1 is the leaf processor, as shown in Figure 1. Processor P_h (where $1 < h < H$) communicates only with P_{h-1} and P_{h+1}. Processor P_H communicates with P_{H-1} and the *Server*. Processor P_1 communicates with P_2 and the *Server*.

6.1 The Top-Down Algorithm

When processor P_h receives a *search(k,x)* message (search for key k using node x) it selects y, the appropriate child of x, and sends the message *search(k,y)* to processor P_{h-1}. When processor P_1 receives a *search(k,x)* message it searches the keys stored in node x for the key k and returns the result to the *Server* processor.

During the *insert* operation processor P_h receives an *insert_transform(k,x)* (transform node x, inserting key k) message. A transformation is applied if x is a 2^B-branch node. In this case a new node x' with splitting key k' is created, and the message *insert_reply(k',x')* is sent to processor P_{h+1}. Oth-

erwise, the message *insert_reply(k,nil)* is sent to processor P_{h+1}, where *nil* is a dummy node value to indicate that no transformation occurred. Processor P_h then selects the appropriate child y and the message *insert_transform(k,y)* is sent to processor P_{h-1}. Processor P_h then waits until it receives the *insert_reply(k",y')* message from P_{h-1} whereupon if y' is not *nil* it adds the new child and splitting key to the node x (or to x' if appropriate). When processor P_1 receives an *insert_transform(k,x)* message it determines whether a transformation should be applied and proceeds in the manner of processor P_h. If the key is not already present it is inserted into the appropriate leaf node. When the *Server* processor receives the *insert_reply(k',y')* messages from processor P_H and y is not *nil*, a new thread is started on processor P_{H+1}, which becomes the new root processor. Figure 2 shows an example of an *insert* operation applied to a $(2,8)$ tree.

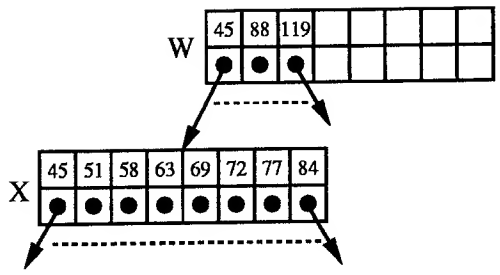
The *delete* operation proceeds in the style of *insert* with the appropriate transformation and replies for updating index information being applied at every stage. However, processor P_{H-1} controls shrinking (as opposed to P_H for growing). Therefore, in the case where a *delete_transform* message is sent to a 2-branch root node at P_H the reply to the *Server* is delayed until P_{H-1} completes communication with P_{H-2}.

Examples of *delete* operations with search key 50 applied to the nodes of a $(2,8)$ tree are shown in Figures 3 and 4. Note that the node address included in the *delete_reply* message indicates the transformation that has been applied. If the address of the transformed node (X in this case) is included in the reply this indicates that a merging transformation was applied. A redistribution occurred if the address of the neighboring node is included.

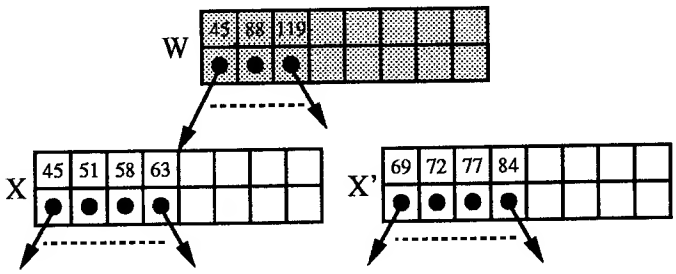
Thus, the top-down algorithm is a lock-coupling algorithm; processor P_h is locked while a transformation is applied at processor P_{h-1}. The reply message causes the lock on P_h to be released and is mandatory following the sending of a transform message to P_{h-1} even if no actual transformation occurs. For n update operations on a tree with approximately n entries, the top-down algorithm requires approximately $n \log_B n$ downward messages and the same number of upward messages.

6.2 The Bottom-Up Algorithm

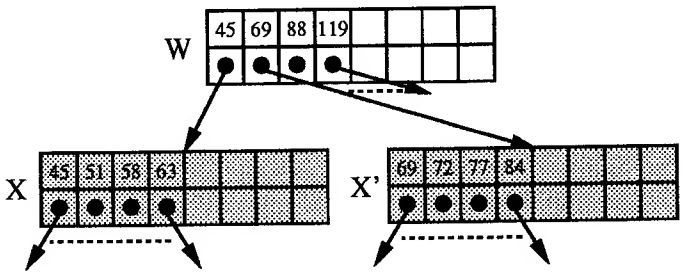
In the implementation of the bottom-up algorithm, two forms of the *find(k,x)* message are used to distinguish between the implementation of a *search* operation (*find_search*) and the implementations of the *insert* and *delete* operations (*find_update*). When processor P_h receives a *find_search(k,x,f)* message, meaning it should search for key k from node x and parent f, it executes *x':=find_node(k,x)* with the following code :



Stage 1: The processor storing W sends an *"insert_transform at node X with key 50"* message to the processor storing X.

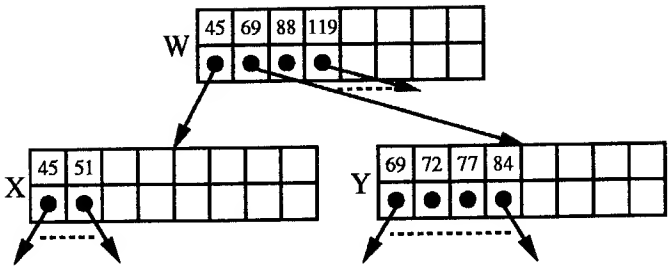


Stage 2: X is split to form node X' and the processor storing X sends an *insert_reply(69,X')* message to the processor storing W.

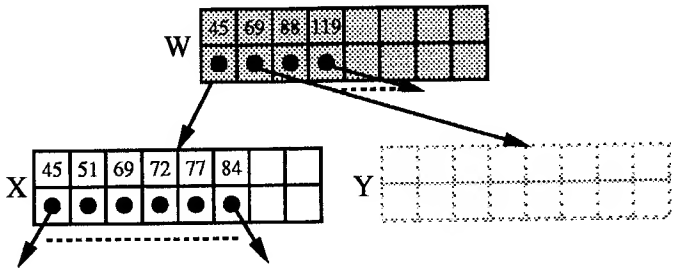


Stage 3: The processor storing W adds the new splitting key, *69*, and the address of X' to W.

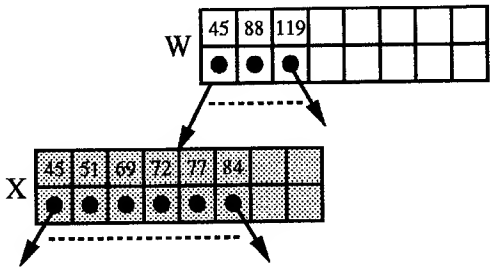
Figure 2: An *insert* operation applied to a $(2,8)$ tree using the top-down algorithm. The shaded nodes represent those nodes that are unchanged during a particular stage in the transformation process.



Stage 1: The processor storing W sends an *"delete_transform at node X using neighbor Y with key 50"* message to the processor storing X.

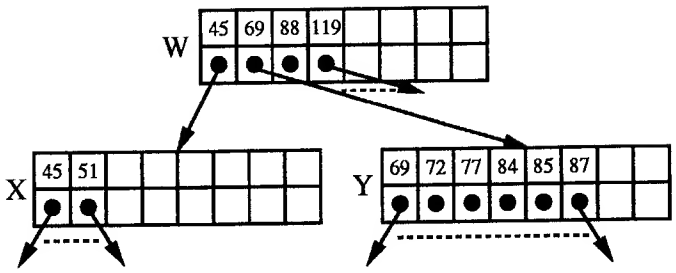


Stage 2: X and Y are merged and Y is deleted. The processor storing X sends an *delete_reply(45,X)* message to the processor storing W.

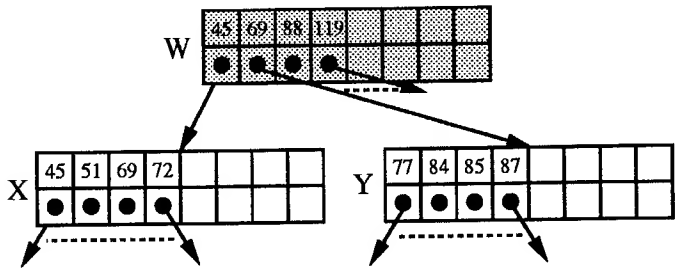


Stage 3: The processor storing W removes the entry for Y from W.

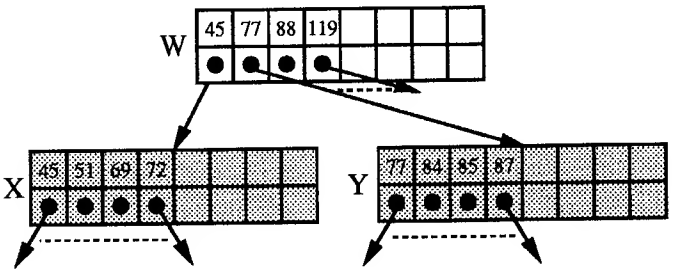
Figure 3: A *delete* operation applied to a $(2,8)$ tree using the top-down algorithm and causing merging.



Stage 1: The processor storing W sends an *"delete_transform at node X using neighbor Y with key 50"* message to the processor storing X.



Stage 2: The children of X and Y are redistributed. The processor storing X sends an *delete_reply(77, Y)* message to the processor storing W.



Stage 3: The processor storing W changes the key value associated with Y.

Figure 4: A *delete* operation applied to a $(2, 8)$ tree using the top-down algorithm and causing redistribution.

```
find_node = procedure(k:key; x:node) returns (node)
    if (x.deleted) then
        if (x.left = nil) then return find_node(k,x.right);
        else return find_node(k,x.left);
    else if (x.i > k) then return find_node(k,x.left);
    else if (x.right = nil) then return(x);
    else if (x.right.i <= k) then return find_node(k,x.right);
    else return(x);
end find_node
```

where *x.right* and *x.left* are the right and left brothers of x (pointed to by the links at x), and *x.deleted* has the value `true` when x has been deleted. *find_node(x,k)* returns the node at the same level as x from which the key k is covered. The node splitting associated with the insertion transformation may cause traversal along the right link and the node redistribution associated with the deletion transformation may cause traversal along the left link.

The additional parameter f of *find_search* is the address of the parent of x. Each search tree node (other than the root) maintains the address of its parent so as to direct the propagation of transformations to higher levels in the search tree. When a *find_search* message is received, the parent pointer of x is assigned the value of f. This propagates the changes made to parents to their children.

Processor P_h then selects y, the appropriate child of x', and sends the message *find_search(k,y,x')* to P_{h-1}. When processor P_1 receives a *find_search(k,x,f)* message it executes *x'=find_node(k,x)*, searches the keys stored in node x' for the key k, and returns the result to the *Server* processor.

During an *insert* or *delete* operation processor P_h receives a *find_update(k,x,f)* message. The routine *find_node* is again called and the appropriate child y of x' is selected. Processor P_h then sends the message *find_update(k,y,x')* to processor P_{h-1}. When processor P_1 receives a *find_update(k,x,f)* message the parent pointer is updated as before and *x'=find_node(k,x)* is executed. A transformation is then applied to x' if required and the key k is either added to (for *insert*) or deleted from (for *delete*) the appropriate node. A message indicating completion is then sent to the *Server* processor.

If an insertion transformation occurred at the leaf level, processor P_1 sends an *insert_transform(k',x',f)* message (insert the new node x' with splitting key k' and parent f) to processor P_2. Processor P_2 calls *find_node(f,k')* to determine the node to which x' and k' should be added. If a transformation occurs as a result of this addition then another *insert_transform* message is sent to processor P_3. This process continues until a level is reached where no transformation occurs. Should the *Server* receive an *insert_transform* message from the root processor then a new thread is started on processor P_{H+1} and P_{H+1} becomes the new root processor. An example of an *insert_transform* message applied to the nodes

of a $(2, 8)$ tree is shown in Figure 5. When the search tree is in Stage 2 of Figure 5, a *find* or *find_update* operation sent to X from W with a key value greater than or equal to *69* results in Z being selected by *find_node*.

The processing of the transformations for the *delete* operation proceeds in the style of *insert* with the appropriate transformation for updating index information being applied at every stage. Examples of *delete_transform* message applied to the nodes of a $(2, 8)$ tree are shown in Figures 6 and 7. Note that the number of key values included in the *delete_transform* message indicates the transformation that was applied at the sender. If only a single key is included then the entry for the child with this key should be removed at the higher level. If two key values are included then the entry for the child with the first value should be changed to the second key value. The use of *find_node* to determine the correct node to update allows the maintenance of the parent pointers to be carried out in this lazy style.

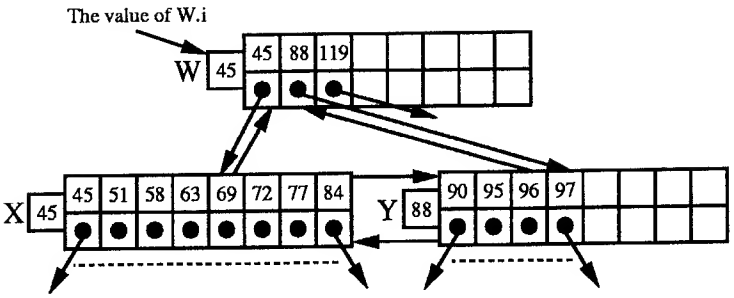
For n update operations the bottom-up algorithm requires approximately $n \log_B n$ downward messages. However, upward messages only occur following a transformation. In practice this results in significantly fewer upward messages than in the top-down case as shown by the experimental results reported in Table 1 in the next section.

7 Relative Performance

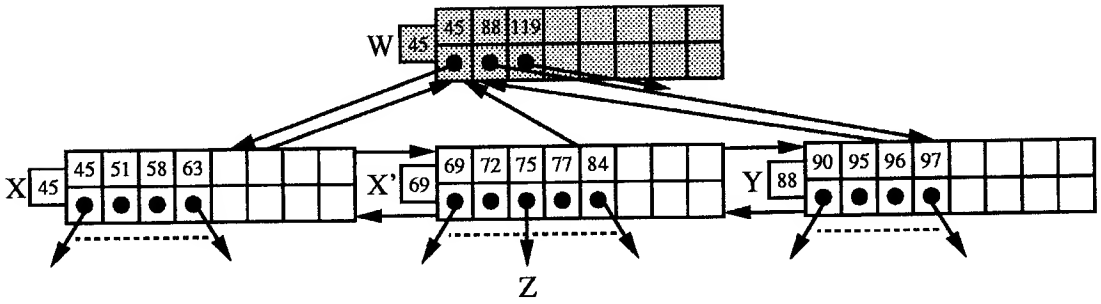
This section compares the relative performance of the two algorithms using the results of simulations. Three sets of simulations measured the throughputs and response times of each algorithm for different values of the branching factor constant B and various query mixes. A fourth set of simulations then measured the relative performance of the algorithms under changes to the one-way message delay. Finally, we compared the performance of the algorithms using synchronous and asynchronous message passing between the processors.

We conducted a number of sets of simulations for each of the algorithms using a range of values for the branching constant B in each case; three sets of results are presented in this paper. B was varied between 2 (this was not strictly a $(2^{B-2}, 2^B)$ search tree and actually represented the 2-3-4 tree used by Carey and Thompson [CT84]) and 8 (a 64-256 tree). The size of the processor array was varied for each case so that only the number of processors required was used. The throughput was measured in terms of the average number of tree operations that complete for every one hundred thousand machine cycles and the response time was measured in terms of the average number of machine cycles between the *Server* sending a query and receiving the corresponding reply.

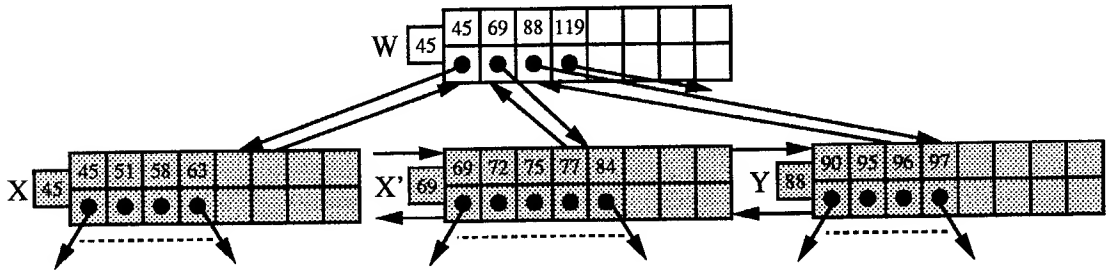
The first set of simulations, *Test 1*, measured the performance of the algorithms when 50,000 *insert* operations using random key values were applied to an initially empty tree. For the second and third



Stage 1: The processor storing X receives an *insert_transform* message informing it that a new node Z has been generated at the level below.

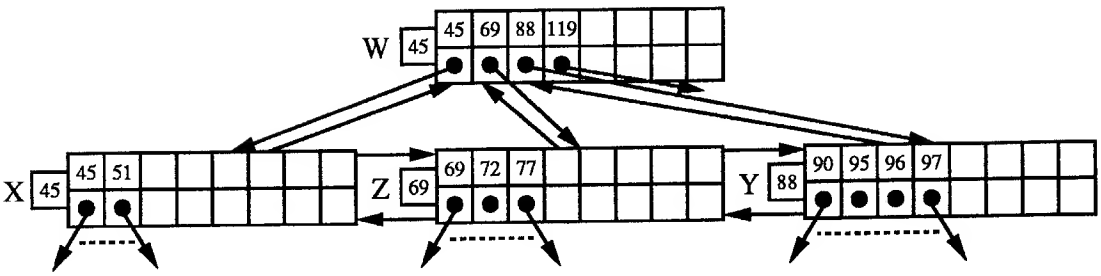


Stage 2: *find_node(75,X)* is invoked, which returns node X in this case. X is split to form node X' and the horizontal links in X and Y are updated accordingly. The processor storing X sends an *insert_transform(69,X',W)* message to the processor storing W.

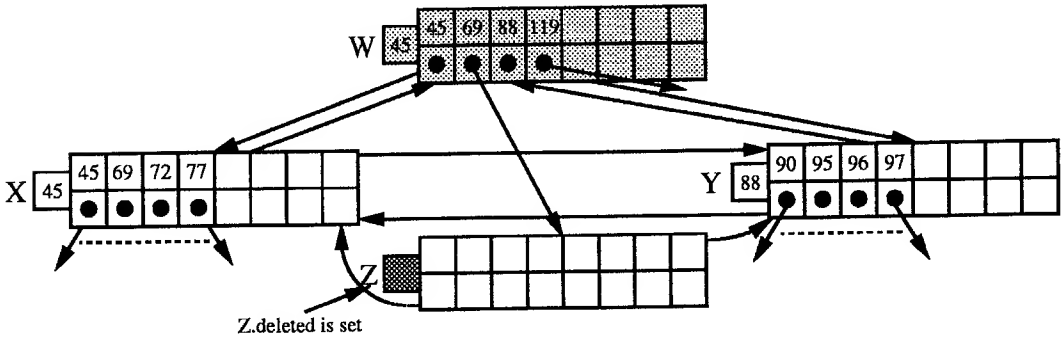


Stage 3: The processor storing W invokes *find_node(69,W)*, which returns node W in this case. The new splitting key, *69*, and the address of X' are added to W. Since no further transformation is required no additional upward messages are generated.

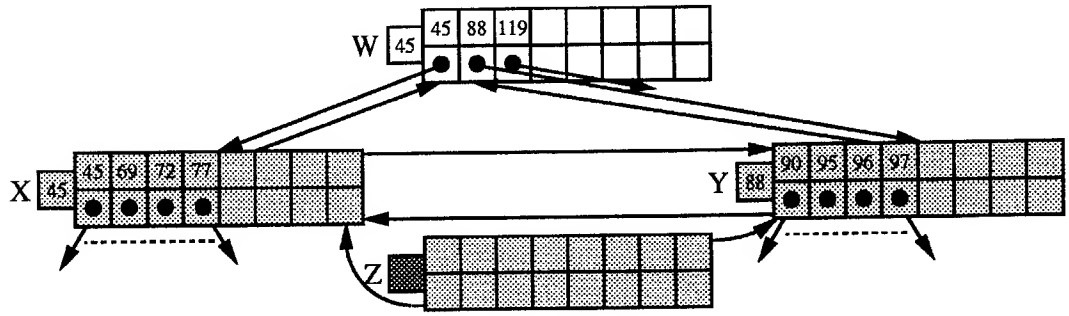
Figure 5: An *insert_transform* operation applied to a $(2,8)$ tree using the bottom-up algorithm. The shaded nodes represent those nodes that are unchanged during a particular stage in the transformation process.



Stage 1: The processor storing X receives a *delete_transform(51,X)* message informing it that the entry for key value *51* has been deleted.

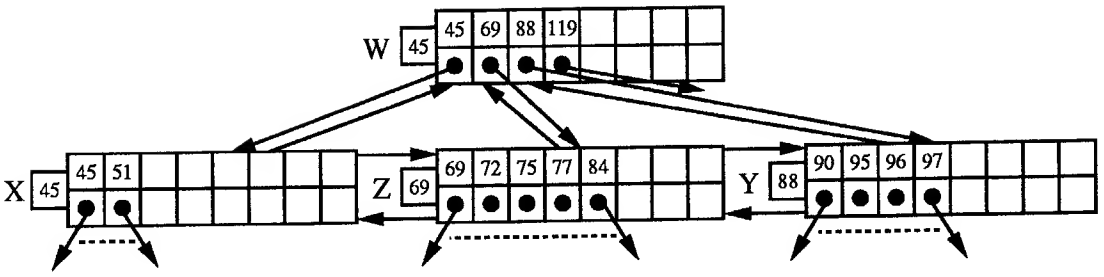


Stage 2: *find_node(75,X)* is invoked, which returns node X in this case. X and Z are merged and Z is marked as deleted. The horizontal links in X and Y are updated accordingly. The processor storing X sends a *delete_transform(69, W)* message to the processor storing W.

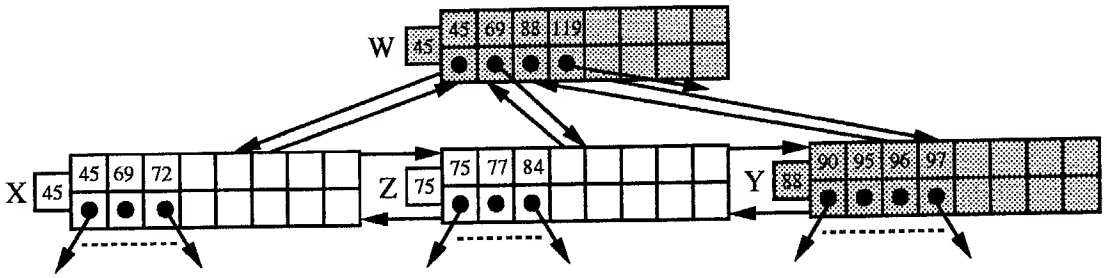


Stage 3: The processor storing W invokes *find_node(69, W)*, which returns node W in this case. The entry for splitting key *69* is removed from W. Since no further transformation is required no additional upward messages are generated.

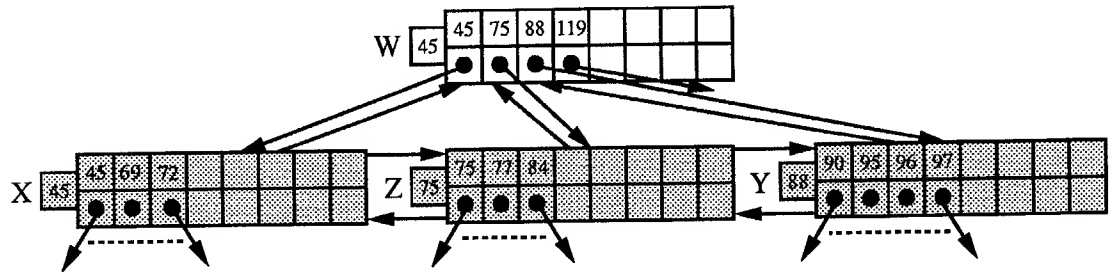
Figure 6: A *delete_transform* operation applied to a $(2, 8)$ tree using the bottom-up algorithm and causing merging.



Stage 1: The processor storing X receives a *delete_transform(51,X)* message informing it that the entry for key value *51* has been deleted.



Stage 2: *find_node(51,X)* is invoked, and returns node X in this case. The children of X and Z are redistributed. The processor storing X sends a *delete_transform(69,75,W)* message to the processor storing W.



Stage 3: The processor storing W invokes *find_node(69,W)*, which returns node W in this case. Its splitting key values are updated. Since no further transformation is required no additional upward messages are generated.

Figure 7: A *delete_transform* operation applied to a $(2, 8)$ tree using the bottom-up algorithm and causing redistribution.

sets of simulations, *Test 2* and *Test 3*, 1,000 *insert* operations using random keys were first applied to an empty search tree followed by 10,000 randomly selected operations. For *Test 2* the percentages of *insert*, *delete* and *search* operations in this random selection were 50%, 30% and 20%, respectively, and for *Test 3* they were 33%, 33% and 34%, respectively. Each time a *delete* operation was applied the key value closest to the selected key value was deleted.

The throughputs and response times are shown in Figures 8 and 9. In all cases the throughput improves for increasing values of B up to 4 (or 5 for *Test 2* applied to the bottom-up algorithm). This peaked response is caused by a trade-off between the number of transformations and the processing time for searching the key values stored at a node. For low values of B, *insert* and *delete* operations cause more transformations to the tree structure. Transformations increase the average processing time for a query and leads to a reduction in throughput. For higher values of B the time required to search and update the keys stored at a node increases. This also increases the average processing time for a query and leads to a reduction in throughput. Therefore an optimal value for B exists where neither effect leads to a significant degradation in throughput.

The response times reach a minimum value as B increases and then grows as B continues to increase. The value of B governs the number of processors in the ring; as B increases the number of processors decreases since a greater number of key values are stored in each node. For increasing values of B below this minimum, the improvement in response time is caused by the reduction in the number of inter-processor hops required for a single query. For increasing values of B greater than the minimum, the degradation in response time occurs due to the increase in the processing time at a single node. Therefore there is a trade-off between the computation at a processor and communication between processors to achieve the optimal response time.

When the two algorithms are compared, the bottom-up algorithm performs better in both throughput and response time for all cases. The improvement in throughput arises because no lock coupling is required and upward messages only occur when a transformation is required. The majority of upward messages in the top-down case merely verify that no transformation took place. The counts of upward messages required during *Test 2* are given in Table 1. The bottom-up algorithm requires significantly fewer upward messages in all cases, leading to very little contention between messages. The numbers of downward messages required by the two algorithms are approximately the same and are equal to the number of upward messages in the top-down case.

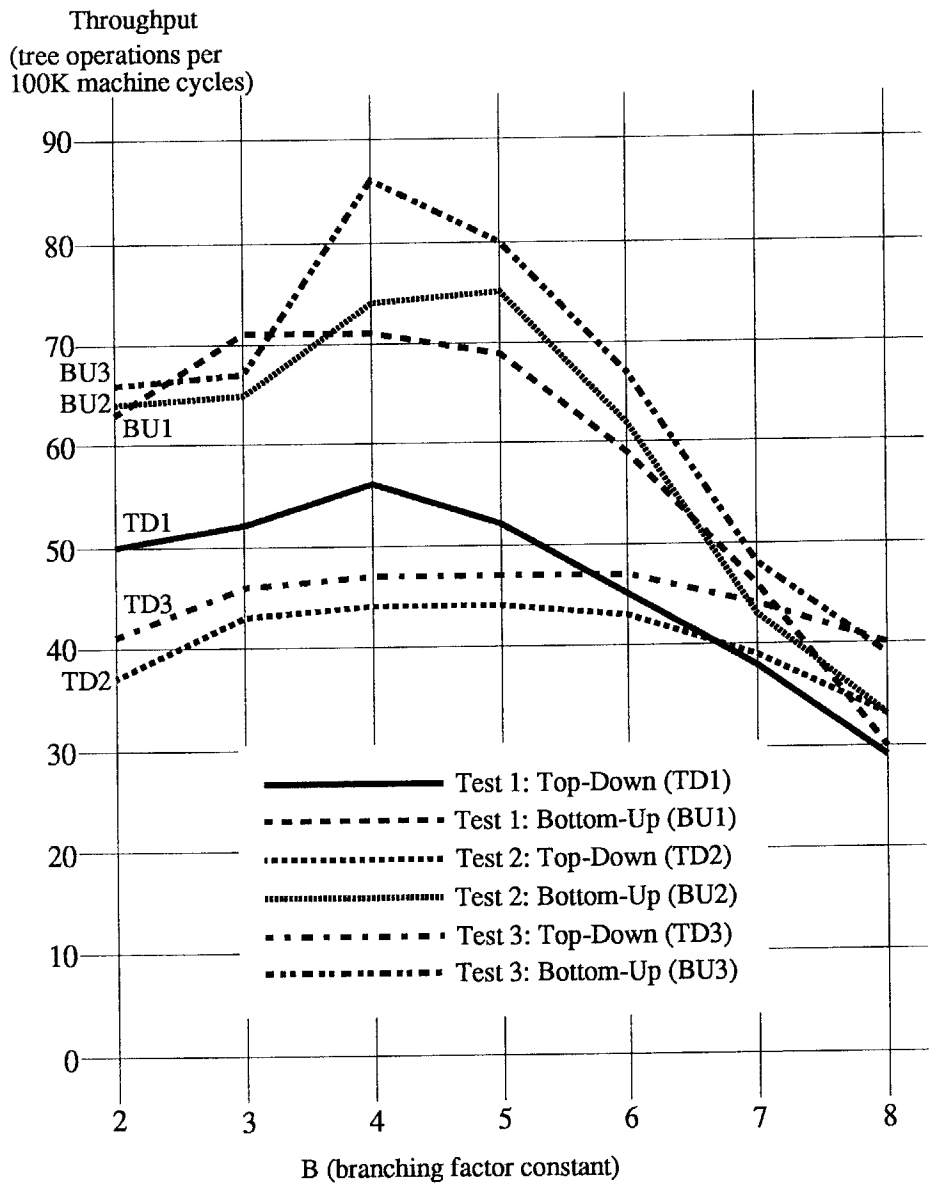


Figure 8: The throughputs for the search trees.

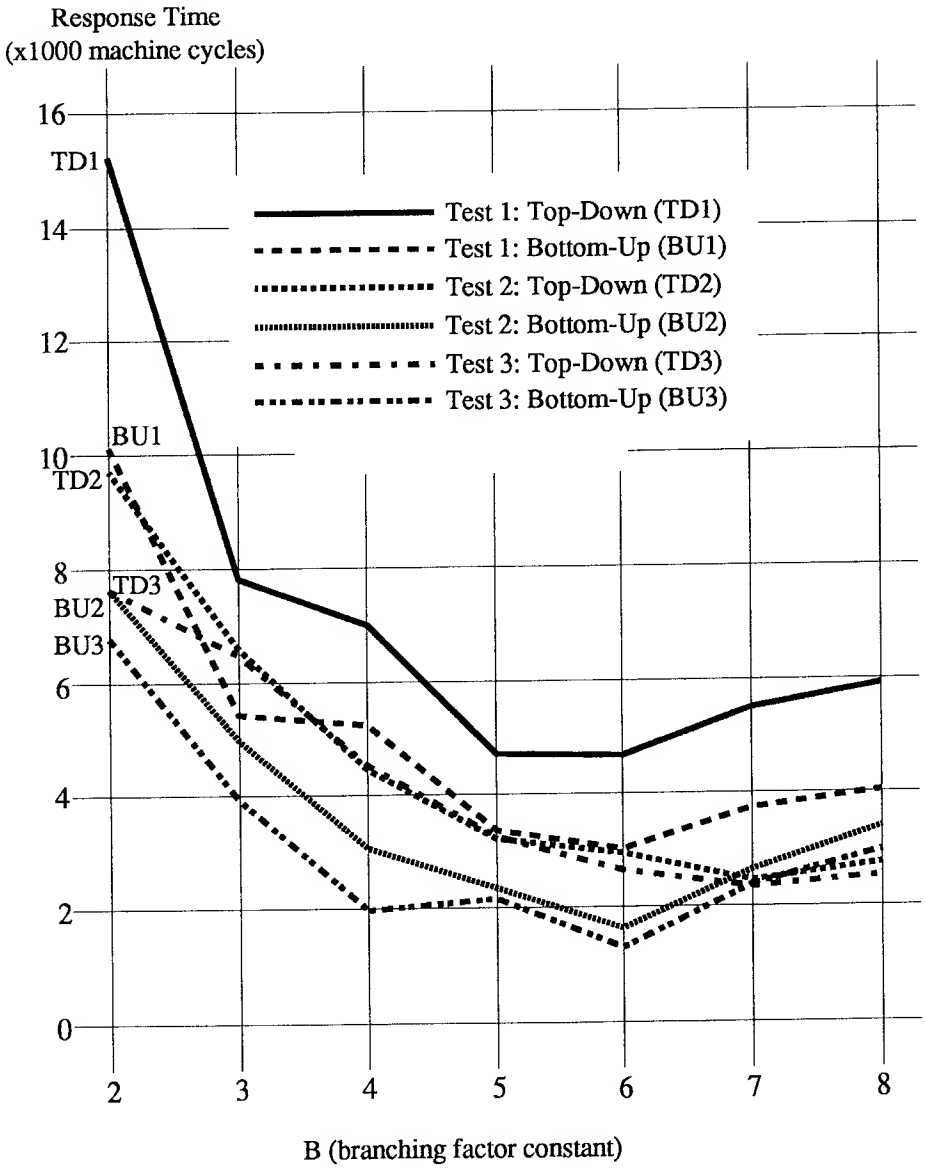


Figure 9: The response times for the search trees.

Branching constant B	Top-Down algorithm	Bottom-Up algorithm
2	65213	2556
3	48805	884
4	37329	345
5	27568	150
6	26143	71
7	19654	36
8	19422	15

Table 1: The numbers of upward messages required during *Test 2*

As noted earlier, the changes to non-leaf levels caused by transformations in the bottom-up algorithm are made by a background task. The *find* operation guarantees that the correct leaf node is reached independent of whether all the required changes have been made. This may require traversal of the horizontal links connecting adjacent nodes leading to an increase in processing time. The numbers of times a horizontal link was traversed during Test 2 are given in Table 2. The number of traversals increases as the value of B decreases but in all cases the numbers are small when compared with the total number of operations, which is 11000. The fact that so few horizontal traversals are made partly accounts for the low response times of the bottom-up algorithm.

Branching constant	Horizontal Traversals
2	29
3	19
4	9
5	7
6	5
7	2
8	2

Table 2: The numbers of horizontal traversals required during *Test 2*

The response time for an individual query is also better in the bottom-up algorithm because transformations at non-leaf levels are conducted after the query has terminated. Transformations propagate up the tree only as far as necessary, which reduces the contention experienced by later queries.

The effect of variations in the message delay was measured for each algorithm when 10,000 *insert* operations using random key values were applied to an (8,32) tree (B=5) using synchronous message

passing. As the one-way message delay increased the throughputs of both the top-down and bottom-up algorithms decreased at approximately the same rate (although the overall percentage change is smaller for the bottom-up algorithm). The response time for both algorithms increased as the one-way message delay increased. However, the increase for the bottom-up algorithm was less significant than that for the top-down algorithm. This difference is caused by the lock coupling and the greater number of upward messages in the top-down case.

Experiments were conducted using both synchronous and asynchronous message passing between processors. The synchronous message style is similar to that used in transputer systems [Inm86]. The synchronous messages have the following protocol. The sender sends a message to the target and waits for an acknowledgment. The message causes an interrupt at the target. The target processor must independently issue a receive command. When a receive command is issued, if a message is already present, the receive routine sends an acknowledgment to the sender and returns the address of the message. If no message has been received, the receive routine waits for the interrupt and then handles the message.

For asynchronous messages the sender sends a message to the target and does not wait for an acknowledgment. The message causes an interrupt at the target and is stored in a buffer of waiting messages. The messages in this buffer are serviced in a FIFO order. When the target processor issues a receive command a waiting message is removed from the buffer. If no messages have been received the receive routine waits for the interrupt. Using asynchronous message as opposed to synchronous messages for the bottom-up algorithm leads to improvements in both throughput and response time as the need for synchronization between adjacent processors is removed and the number of messages between processors decreases. However, the top-down algorithm is implicitly synchronous since the parent processor always waits for the reply from its son. Thus moving from synchronous to asynchronous message passing does not affect performance in this case.

8 Design Alternatives

In the B-tree algorithms described in [WW90, Wan91, LY81] the rightmost node examined at each level is pushed onto a stack during the downward traversal of the search tree. This stack is included in the *find_update* message. If a transformation occurs, the contents of the stack are used to provide an indication of the node to be updated at a given level in the tree. This is only an indication as subsequent transformations may have occurred between the original downward traversal and the propagation of transformations. The message length for update operations and transformations is $\mathcal{O}(\log n)$, since the stack size, and hence the message size, is proportional to the height of the tree.

The alternative technique used in the algorithm described here maintains a pointer to the parent node at each node (other than the root) in the search tree. These pointers are used to give a similar indication of the node to be updated following a transformation. A question arises as to how the pointers should be maintained as transformations at level h cause inconsistencies in some of the parent pointers in level *h-1*. We investigated several solutions and compared their performance to that of the stack-based method.

The approach described in Section 5.2 includes the address of the parent node in every *find* message sent to a child regardless of whether a change is required. We have termed this a conservative pointer-based approach. The address is then assigned to the parent pointer of the child. This leads to a shorter average message length than the stack-based method ($\mathcal{O}(1)$ compared to $\mathcal{O}(\log n)$), and proved to be simpler to implement. In addition, since the original downward traversal will have updated the parent pointers of all the nodes accessed, the indication given by the pointer-based approach will always be at least as good as that given by the stack-based method. In cases where changes to the parent pointers occur between the original downward traversal and the propagation of the transformations, the indication given by the pointer-based approach is better than that given by the stack-based method. Although the differences in performance are marginal, for long processor arrays, where the messages in the stack-based method are significantly longer, the pointer-based approach has superior performance. In systems where messages are divided into small packets before transmission across the network, the address-based approach may lead to fewer packets, therefore reducing network latency and contention and improving performance.

Two alternative techniques for maintaining the parent pointers were investigated. Maintaining a list of the inconsistent children at level h and notifying these nodes of the change to their parent pointer when they are next accessed proves to be a costly solution. Checks have to be made before sending a message and after receiving a message to determine whether changes are required. Alternatively, a new message type may be introduced that is sent by processor P_h to processor P_{h-1} following a transformation at P_h. The message simply notifies P_{h-1} of the required changes to the parent pointers. Although this approach has shorter messages than the conservative pointer-based approach, it leads to an increase in the total number of messages. Both these alternatives exhibit poorer performance than the stack-based and the conservative pointer-based approaches.

Several different algorithms can be used during the execution of the *find* operation to ensure the required leaf node is reached in the bottom-up algorithm. The small number of horizontal traversals given in Table 2 for the bottom-up algorithm suggests that choosing the wrong node during a downward traversal is a very infrequent event. A possible optimization of the algorithm is to assume that the correct node is always chosen and to only execute the *find_node* routine given in Section 6.2 at the leaf

level. If the wrong node is chosen at a branch level then the downward traversal will descend to either the leftmost or rightmost node in the subtree of which the chosen node is the root – leftmost if the chosen node has a key value greater than the search key and rightmost if the chosen node has a key value less than the search key. This gives about a 10% improvement in response time over the *find_node* algorithm given in Section 6.2. The throughputs are approximately the same; the execution of *find_node* at the leaf level prevents any improvement. However, an optimization can be made so that *find_node* is called at every level only when the wrong node may have been selected. If the leftmost or rightmost child is selected as the next node then *find_node* is called and any required horizontal traversals are made. This leads to similar improvements in response time over the algorithm given in Section 6.2.

9 Conclusions

We have shown that a $(2^{B-2}, 2^B)$ search tree of n entries can be implemented on a linear array of up to $\lceil \log_2 n/(B-2) \rceil + 1$ processors, where each processor stores a level of the tree structure. Such a linear array may be physically mapped onto processors in two or three dimensions on the majority of available architectures. Updates can be performed on the tree using both top-down and bottom-up algorithms.

The top-down node-splitting algorithm uses lock coupling to apply transformations during a single traversal of the tree structure. However, processors are required to wait for replies most of which merely verify that no transformation occurred.

The introduction of side links between adjacent nodes at the same level eliminates the need for lock coupling and permits a bottom-up algorithm to be used. This algorithm allows the transformations resulting from changes to the tree structure to be performed asynchronously from the leaf nodes upwards, while guaranteeing the correctness of other operations concurrently executing on the data structure. The use of parent pointers to give an indication of the node to be updated at a higher level leads to improved performance when compared to the stack-based technique used in other B-tree algorithms.

In a series of simulations conducted for both algorithms, the bottom-up approach gives significantly better query throughput and response time. The number of upward messages (and hence the contention) between adjacent processors in the linear array is significantly less for the bottom-up algorithm. Furthermore, the bottom-up algorithm shows increasingly superior performance relative to the top-down algorithm as the one-way message delay between adjacent processors increases. Improved performance is also achieved for the bottom-up algorithm when asynchronous as opposed to synchronous message passing is used.

The bottom-up algorithm for the $(2^{B-2}, 2^B)$ search tree has been shown to provide a highly efficient and flexible implementation of dictionary abstract data types on message-passing MIMD architectures.

For a given cost ratio of computation to communication the value of B may be varied to maximize performance. The algorithm also introduces a stabilizing hysteresis behavior that is not present in many other balanced-tree algorithms. The bottom-up algorithm described here has the best performance of any of the implementations of search trees for linear arrays described to date.

10 Acknowledgments

We thank Anant Agarwal, Wilson Hsieh, Anthony Joseph and Sharon Perl for their comments and suggestions on this work.

References

[Bay72] R. Bayer. Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.

[BDCW91] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, 1991.

[Bre91] E.A. Brewer. Aspects of a high-performance parallel-architecture simulator. Master's thesis, MIT Laboratory for Computer Science, 1991.

[Com79] D. Comer. The ubiquitous B-tree. *Computer Surveys*, 11(2):121–137, 1979.

[CS90] A. Colbrook and C. Smythe. Efficient implementation of search trees on parallel distributed memory architectures. *IEE Proceedings Part E*, 137:394–400, 1990.

[CT84] M.J. Carey and C.D. Thompson. An efficient implementation of search trees on [lgn+1] processors. *IEEE Transactions on Computers*, 33(11):1038–1041, 1984.

[Del91] C.N. Dellarocas. A high-performance retargetable simulator for parallel architectures. Master's thesis, MIT Laboratory for Computer Science, 1991.

[Fis84] A.L. Fisher. Dictionary machines with a small number of processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 151–156, 1984.

[GS78] L.J. Guibas and R. Sedgewick. A dichromatic framework for balancing trees. In *Proceedings of the 19th Annual IEEE Computer Society Symposium on the Foundations of Computer Science*, pages 8–21, 1978.

[Inm86] Inmos. *Transputer Reference Manual.* Prentice Hall, London, 1986.

[LSS87] V. Lanin, D. Shasha, and J. Schmidt. An analytical model for the performance of concurrent B-tree algorithms. Technical report, Ultracomputer Laboratory, New York University, 1987.

[LY81] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[MR85] Y. Mond and Y. Raz. Concurrency control in B+-trees using preparatory operations. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 331–334, 1985.

[Qui87] M.J. Quinn. *Designing efficient algorithms for parallel computers.* McGraw-Hill, New York, 1987.

[Sag86] Y. Sagiv. Concurrent operations on B-trees with overtaking. *Journal of Computer and System Sciences*, 33(2):275–296, 1986.

[SG88] D. Shasha and N. Goodman. Concurrent search tree algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, 1988.

[TNM80] Y. Tanaka, Y. Nozaka, and A. Masuyama. Pipeline searching and sorting modules as components of a data flow database computer. In *Proceedings of the International Federation for Information Processing*, pages 427–432. North-Holland, 1980.

[Wan91] P. Wang. An in-depth analysis of concurrent B-tree algorithms. Master's thesis, MIT Laboratory for Computer Science, 1991.

[WW90] W.E. Weihl and P. Wang. Multi-version memory: Software cache management for concurrent B-trees. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 650–655, 1990.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR 517	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-89-J-1988
6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS (*City, State, and ZIP Code*) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (*City, State, and ZIP Code*) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)

Algorithms for Search Trees on Message-Passing Architectures

12. PERSONAL AUTHOR(S)
Adrian Colbrook, Eric A. Brewer, Chrysanthos N. Dellarocas, William E. Weihl

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	September 1991	27

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Balanced search trees, parallel algorithms, linear processor array, message-passing architectures.

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

In this paper we describe a new algorithm for maintaining a balanced search tree on a message-passing MIMD architecture; the algorithm is particularly well suited for implementation on a small number of processors. We introduce a $(2^{B-2}, 2^B)$ search tree that uses a linear array of $\mathcal{O}(\log n)$ processors to store n entries. Update operations use a bottom-up node-splitting scheme, which performs better than top-down search tree algorithms. Additionally, for a given cost ratio of computation to communication the value of B may be varied to maximize performance. Implementations on a parallel-architecture simulator are described.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Carol Nicolora	22b. TELEPHONE (*Include Area Code*) (617) 253-5894 22c. OFFICE SYMBOL